05｜标准先行：Go项目的布局标准是什么？

2021-10-22 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 18:06 大小 16.58M

你好，我是 Tony Bai。

在前面的讲解中，我们编写的 Go 程序都是简单程序，一般由一个或几个 Go 源码文件组成，而且所有源码文件都在同一个目录中。但是生产环境中运行的实用程序可不会这么简单，通常它们都有着复杂的项目结构布局。弄清楚一个实用 Go 项目的项目布局标准是 Go 开发者走向编写复杂 Go 程序的第一步，也是必经的一步。

但 Go 官方到目前为止也没有给出一个关于 Go 项目布局标准的正式定义。那在这样的情况下，Go 社区是否有我们可以遵循的参考布局，或者事实标准呢？我可以肯定的告诉你，有的。在这一节课里，我就来告诉你 Go 社区广泛采用的 Go 项目布局是什么样子的。

要想了解 Go 项目的结构布局以及演化历史，全世界第一个 Go 语言项目是一个最好的切入点。所以，我们就先来看一下 Go 语言“创世项目”的结构布局是什么样的。

Go 语言“创世项目”结构是怎样的？

什么是“**Go 语言的创世项目**”呢？其实就是 Go 语言项目自身，它是全世界第一个 Go 语言项目。但这么说也不够精确，因为 Go 语言项目从项目伊始就混杂着多种语言，而且以 C 和 Go 代码为主，Go 语言的早期版本 C 代码的比例还不小。

我们先用 loccount 工具对 Go 语言发布的第一个 Go 1.0 版本分析看看：

复制代码

```
$loccount .
all           SLOC=460992  (100.00%)  LLOC=193045  in 2746 files
Go            SLOC=256321  (55.60%)  LLOC=109763  in 1983 files
C             SLOC=148001  (32.10%)  LLOC=73458   in 368 files
HTML          SLOC=25080   (5.44%)  LLOC=0       in 57 files
asm           SLOC=10109   (2.19%)  LLOC=0       in 133 files
... ...
```

你会发现，在 1.0 版本中，Go 代码行数占据一半以上比例，但是 C 语言代码行数也占据了 32.10% 的份额。而且在后续 Go 版本演进过程中，Go 语言代码行数占比还在逐步提升，直到 Go 1.5 版本实现自举后，Go 语言代码行数占比将近 90%，C 语言比例下降为不到 1%，这一比例一直延续至今。

虽然 C 代码比例下降，Go 代码比例上升，但 Go 语言项目的布局结构却整体保留了下来，十多年间虽然也有一些小范围变动，但整体没有本质变化。作为 Go 语言的“创世项目”，它的结构布局对后续 Go 社区的项目具有重要的参考价值，尤其是 Go 项目早期 src 目录下面的结构。

为了方便查看，我们首先下载 Go 语言创世项目源码：

复制代码

```
$git clone https://github.com/golang/go.git
```

进入 Go 语言项目根目录后，我们使用 tree 命令来查看一下 Go 语言项目自身的最初源码结构布局，以 Go 1.3 版本为例，结果是这样的：

复制代码

```
$cd go // 进入Go语言项目根目录
$git checkout go1.3 // 切换到go 1.3版本
$tree -LF 1 ./src // 查看src目录下的结构布局
./src
├── all.bash*
├── clean.bash*
├── cmd/
├── make.bash*
├── Make.dist
├── pkg/
├── race.bash*
├── run.bash*
... ...
└── sudo.bash*
```

从上面的结果来看，src 目录下面的结构有这三个特点。

首先，你可以看到，以 all.bash 为代表的代码构建的脚本源文件放在了 src 下面的顶层目录下。

第二，src 下的二级目录 cmd 下面存放着 Go 相关可执行文件的相关目录，我们可以深入查看一下 cmd 目录下的结构：

复制代码

```
$ tree -LF 1 ./cmd
./cmd
... ...
├── 6a/
├── 6c/
├── 6g/
... ...
├── cc/
├── cgo/
├── dist/
├── fix/
├── gc/
├── go/
├── gofmt/
├── ld/
```

```
16 ├── nm/
17 ├── objdump/
18 ├── pack/
19 └── yacc/
```

我们可以看到，这里的每个子目录都是一个 Go 工具链命令或子命令对应的可执行文件。其中，6a、6c、6g 等是早期 Go 版本针对特定平台的汇编器、编译器等的特殊命名方式。

第三个特点，你会看到 src 下的二级目录 pkg 下面存放着运行时实现、标准库包实现，这些包既可以被上面 cmd 下各程序所导入，也可以被 Go 语言项目之外的 Go 程序依赖并导入。下面是我们通过 tree 命令查看 pkg 下面结构的输出结果：

复制代码

```
1  # tree -LF 1 ./pkg
2  ./pkg
3  ... ...
4  ├── flag/
5  ├── fmt/
6  ├── go/
7  ├── hash/
8  ├── html/
9  ├── image/
10 ├── index/
11 ├── io/
12 ... ...
13 ├── net/
14 ├── os/
15 ├── path/
16 ├── reflect/
17 ├── regexp/
18 ├── runtime/
19 ├── sort/
20 ├── strconv/
21 ├── strings/
22 ├── sync/
23 ├── syscall/
24 ├── testing/
25 ├── text/
26 ├── time/
27 ├── unicode/
28 └── unsafe/
```

虽然 Go 语言的创世项目的 src 目录下的布局结构，离现在已经比较久远了，但是这样的布局特点依然对后续很多 Go 项目的布局产生了比较大的影响，尤其是那些 Go 语言早期

采纳者建立的 Go 项目。比如，Go 调试器项目 Delve、开启云原生时代的 Go 项目 Docker，以及云原生时代的“操作系统”项目 Kubernetes 等，它们的项目布局，至今都还保持着与 Go 创世项目早期相同的风格。

当然了，这些早期的布局结构一直在不断地演化，简单来说可以归纳为下面三个比较重要的演进。

演进一：Go 1.4 版本删除 pkg 这一中间层目录并引入 internal 目录

出于简化源码树层次的原因，Go 语言项目的 Go 1.4 版本对它原来的 src 目录下的布局做了两处调整。第一处是删除了 Go 源码树中“src/pkg/xxx”中 pkg 这一层级目录而直接使用 src/xxx。这样一来，Go 语言项目的源码树深度减少一层，更便于 Go 开发者阅读和探索 Go 项目源码。

另外一处就是 Go 1.4 引入 internal 包机制，增加了 internal 目录。这个 internal 机制其实是所有 Go 项目都可以用的，Go 语言项目自身也是自 Go 1.4 版本起，就使用 internal 机制了。根据 internal 机制的定义，一个 Go 项目里的 internal 目录下的 Go 包，只可以被本项目内部的包导入。项目外部是无法导入这个 internal 目录下面的包的。可以说，internal 目录的引入，让一个 Go 项目中 Go 包的分类与用途变得更加清晰。

演进二：Go1.6 版本增加 vendor 目录

第二次的演进，其实是为了解决 Go 包依赖版本管理的问题，Go 核心团队在 Go 1.5 版本中做了第一次改进。增加了 vendor 构建机制，也就是 Go 源码的编译可以不在 GOPATH 环境变量下面搜索依赖包的路径，而在 vendor 目录下查找对应的依赖包。

Go 语言项目自身也在 Go 1.6 版本中增加了 vendor 目录以支持 vendor 构建，但 vendor 目录并没有实质性缓存任何第三方包。直到 Go 1.7 版本，Go 才真正在 vendor 下缓存了其依赖的外部包。这些依赖包主要是 golang.org/x 下面的包，这些包同样是由 Go 核心团队维护的，并且其更新速度不受 Go 版本发布周期的影响。

vendor 机制与目录的引入，让 Go 项目第一次具有了可重现构建（Reproducible Build）的能力。

演进三：Go 1.13 版本引入 go.mod 和 go.sum

第三次演进，还是为了解决 Go 包依赖版本管理的问题。在 Go 1.11 版本中，Go 核心团队做出了第二次改进尝试：引入了 Go Module 构建机制，也就是在项目引入 go.mod 以及在 go.mod 中明确项目所依赖的第三方包和版本，项目的构建就将摆脱 GOPATH 的束缚，实现精准的可重现构建。

Go 语言项目自身在 Go 1.13 版本引入 go.mod 和 go.sum 以支持 Go Module 构建机制，下面是 Go 1.13 版本的 go.mod 文件内容：

复制代码

```
module std

go 1.13

require (
  golang.org/x/crypto v0.0.0-20190611184440-5c40567a22f8
  golang.org/x/net v0.0.0-20190813141303-74dc4d7220e7
  golang.org/x/sys v0.0.0-20190529130038-5219a1e1c5f8 // indirect
  golang.org/x/text v0.3.2 // indirect
)
```

我们看到，Go 语言项目自身所依赖的包在 go.mod 中都有对应的信息，而原本这些依赖包是缓存在 vendor 目录下的。

总的来说，这三次演进主要体现在简化结构布局，以及优化包依赖管理方面，起到了改善 Go 开发体验的作用。可以说，Go 创世项目的源码布局以及演化对 Go 社区项目的布局具有重要的启发意义，以至于在多年的 Go 社区实践后，Go 社区逐渐形成了公认的 Go 项目的典型结构布局。

现在的 Go 项目的典型结构布局是怎样的？

一个 Go 项目通常分为可执行程序项目和库项目，现在我们就来分析一下这两类 Go 项目的典型结构布局分别是怎样的。

首先我们先来看 Go 可执行程序项目的典型结构布局。

可执行程序项目是以构建可执行程序为目的的项目，Go 社区针对这类 Go 项目所形成的典型结构布局是这样的：

复制代码

```
$tree -F exe-layout
exe-layout
├── cmd/
│   ├── app1/
│   │   └── main.go
│   └── app2/
│       └── main.go
├── go.mod
├── go.sum
├── internal/
│   ├── pkga/
│   │   └── pkg_a.go
│   └── pkgb/
│       └── pkg_b.go
├── pkg1/
│   └── pkg1.go
├── pkg2/
│   └── pkg2.go
└── vendor/
```

这样的一个 Go 项目典型布局就是“脱胎”于 Go 创世项目的最新结构布局，我现在跟你解释一下这里面的几个要点。

我们从上往下按顺序来，先来看 **cmd 目录**。cmd 目录就是存放项目要编译构建的可执行文件对应的 main 包的源文件。如果你的项目中有多个可执行文件需要构建，每个可执行文件的 main 包单独放在一个子目录中，比如图中的 app1、app2，cmd 目录下的各 app 的 main 包将整个项目的依赖连接在一起。

而且通常来说，main 包应该很简洁。我们在 main 包中会做一些命令行参数解析、资源初始化、日志设施初始化、数据库连接初始化等工作，之后就会将程序的执行权限交给更高级的执行控制对象。另外，也有一些 Go 项目将 cmd 这个名字改为 app 或其他名字，但它的功能其实并没有变。

接着我们来看 **pkgN 目录**，这是一个存放项目自身要使用、同样也是可执行文件对应 main 包所要依赖的库文件，同时这些目录下的包还可以被外部项目引用。

然后是 **go.mod** 和 **go.sum**，它们是 Go 语言包依赖管理使用的配置文件。我们前面说过，Go 1.11 版本引入了 Go Module 构建机制，这里我建议你所有新项目都基于 Go Module 来进行包依赖管理，因为这是目前 Go 官方推荐的标准构建模式。

对于还没有使用 Go Module 进行包依赖管理的遗留项目，比如之前采用 dep、glide 等作为包依赖管理工具的，建议尽快迁移到 Go Module 模式。Go 命令支持直接将 dep 的 Gopkg.toml/Gopkg.lock 或 glide 的 glide.yaml/glide.lock 转换为 go.mod。

最后我们再来看看 **vendor 目录**。vendor 是 Go 1.5 版本引入的用于在项目本地缓存特定版本依赖包的机制，在 Go Modules 机制引入前，基于 vendor 可以实现可重现构建，保证基于同一源码构建出的可执行程序是等价的。

不过呢，我们这里将 vendor 目录视为一个可选目录。原因在于，Go Module 本身就支持可再现构建，而无需使用 vendor。 当然 Go Module 机制也保留了 vendor 目录（通过 go mod vendor 可以生成 vendor 下的依赖包，通过 go build -mod=vendor 可以实现基于 vendor 的构建）。一般我们仅保留项目根目录下的 vendor 目录，否则会造成不必要的依赖选择的复杂性。

当然了，有些开发者喜欢借助一些第三方的构建工具辅助构建，比如：make、bazel 等。你可以将这类外部辅助构建工具涉及的诸多脚本文件（比如 Makefile）放置在项目的顶层目录下，就像 Go 创世项目中的 all.bash 那样。

另外，这里只要说明一下的是，Go 1.11 引入的 module 是一组同属于一个版本管理单元的包的集合。并且 Go 支持在一个项目 / 仓库中存在多个 module，但这种管理方式可能要比一定比例的代码重复引入更多的复杂性。 因此，如果项目结构中存在版本管理的“分歧”，比如：app1 和 app2 的发布版本并不总是同步的，那么我建议你将项目拆分为多个项目（仓库），每个项目单独作为一个 module 进行单独的版本管理和演进。

当然如果你非要在一个代码仓库中存放多个 module，那么新版 Go 命令也提供了很好的支持。比如下面代码仓库 multi-modules 下面有三个 module：mainmodule、module1 和 module2：

复制代码

```
1 $tree multi-modules
2 multi-modules
```

```
├── go.mod // mainmodule
├── module1
│   └── go.mod // module1
└── module2
    └── go.mod // module2
```

我们可以通过 git tag 名字来区分不同 module 的版本。其中 vX.Y.Z 形式的 tag 名字用于代码仓库下的 mainmodule；而 module1/vX.Y.Z 形式的 tag 名字用于指示 module1 的版本；同理，module2/vX.Y.Z 形式的 tag 名字用于指示 module2 版本。

如果 Go 可执行程序项目有一个且只有一个可执行程序要构建，那就比较好办了，我们可以将上面项目布局进行简化：

复制代码

```
$tree -F -L 1 single-exe-layout
single-exe-layout
├── go.mod
├── internal/
├── main.go
├── pkg1/
├── pkg2/
└── vendor/
```

你可以看到，我们删除了 cmd 目录，将唯一的可执行程序的 main 包就放置在项目根目录下，而其他布局元素的功用不变。

好了到这里，我们已经了解了 Go 可执行程序项目的典型布局，现在我们再来看看 Go 库项目的典型结构布局是怎样的。

Go 库项目仅对外暴露 Go 包，这类项目的典型布局形式是这样的：

复制代码

```
$tree -F lib-layout
lib-layout
├── go.mod
├── internal/
│   ├── pkga/
│   │   └── pkg_a.go
│   └── pkgb/
│       └── pkg_b.go
```

```
9  ├── pkg1/
10 │   └── pkg1.go
11 └── pkg2/
12     └── pkg2.go
```

我们看到，库类型项目相比于 Go 可执行程序项目的布局要简单一些。因为这类项目不需要构建可执行程序，所以去除了 cmd 目录。

而且，在这里，vendor 也不再是可选目录了。对于库类型项目而言，我们并不推荐在项目中放置 vendor 目录去缓存库自身的第三方依赖，库项目仅通过 go.mod 文件明确表述出该项目依赖的 module 或包以及版本要求就可以了。

Go 库项目的初衷是为了对外部（开源或组织内部公开）暴露 API，对于仅限项目内部使用而不想暴露到外部的包，可以放在项目顶层的 internal 目录下面。当然 internal 也可以有多个并存在于项目结构中的任一目录层级中，关键是项目结构设计人员要明确各级 internal 包的应用层次和范围。

对于有一个且仅有一个包的 Go 库项目来说，我们也可以将上面的布局做进一步简化，简化的布局如下所示：

复制代码

```
1 $tree -L 1 -F single-pkg-lib-layout
2 single-pkg-lib-layout
3 ├── feature1.go
4 ├── feature2.go
5 ├── go.mod
6 └── internal/
```

简化后，我们将这唯一包的所有源文件放置在项目的顶层目录下（比如上面的 feature1.go 和 feature2.go），其他布局元素位置和功用不变。

好了，现在我们已经了解完目前 Go 项目的典型结构布局了。不过呢，除了这些之外，还要注意一下早期 Go 可执行程序项目的经典布局，这个又有所不同。

注意早期 Go 可执行程序项目的典型布局

很多早期接纳 Go 语言的开发者所建立的 Go 可执行程序项目，深受 Go 创世项目 1.4 版本之前的布局影响，这些项目将所有可暴露到外面的 Go 包聚合在 pkg 目录下，就像前面 Go 1.3 版本中的布局那样，它们的典型布局结构是这样的：

复制代码

```
$tree -L 3 -F early-project-layout
early-project-layout
└── exe-layout/
    ├── cmd/
    │   ├── app1/
    │   └── app2/
    ├── go.mod
    ├── internal/
    │   ├── pkga/
    │   └── pkgb/
    ├── pkg/
    │   ├── pkg1/
    │   └── pkg2/
    └── vendor/
```

我们看到，原本放在项目顶层目录下的 pkg1 和 pkg2 公共包被统一聚合到 pkg 目录下了。而且，这种早期 Go 可执行程序项目的典型布局在 Go 社区内部也不乏受众，很多新建的 Go 项目依然采用这样的项目布局。

所以，当你看到这样的布局也不要奇怪，并且在我的讲解后，你应该就明确在这样的布局下 pkg 目录所起到的“聚类”的作用了。不过，在这里还是建议你在创建新的 Go 项目时，优先采用前面的标准项目布局。

小结

到这里，我们今天这门课就结束了。在这一节课里，我们学习了 Go 创世项目，也就是 Go 语言项目自身的项目源码布局，以及演进情况。在 Go 创世项目的启发下，Go 社区在多年实践中形成了典型的 Go 项目结构布局形式。

我们将 Go 项目分为可执行程序项目和 Go 库项目两类进行了详细的项目典型布局讲解，这里简单回顾一下。

首先，对于以生产可执行程序为目的的 Go 项目，它的典型项目结构分为五部分：

放在项目顶层的 Go Module 相关文件，包括 go.mod 和 go.sum；

cmd 目录：存放项目要编译构建的可执行文件所对应的 main 包的源码文件；

项目包目录：每个项目下的非 main 包都“平铺”在项目的根目录下，每个目录对应一个 Go 包；

internal 目录：存放仅项目内部引用的 Go 包，这些包无法被项目之外引用；

vendor 目录：这是一个可选目录，为了兼容 Go 1.5 引入的 vendor 构建模式而存在的。这个目录下的内容均由 Go 命令自动维护，不需要开发者手工干预。

第二，对于以生产可复用库为目的的 Go 项目，它的典型结构则要简单许多，我们可以直接理解为在 Go 可执行程序项目的基础上去掉 cmd 目录和 vendor 目录。

最后，早期接纳 Go 语言的开发者所建立的项目的布局深受 Go 创世项目 1.4 版本之前布局的影响，将可导出的公共包放入单独的 pkg 目录下，我们了解这种情况即可。对于新建 Go 项目，我依旧建议你采用前面介绍的标准布局形式。

现在，如果你要再面对一个要用于生产环境的 Go 应用项目的布局问题，是不是胸有成竹了呢？

思考题

如果非要你考虑 Go 项目结构的最小标准布局，那么你觉得这个布局中都应该包含哪些东西呢？欢迎在留言区留下你的答案。

感谢你和我一起学习，也欢迎你把这节课分享给更多对 Go 项目布局感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 8 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04｜初窥门径：一个Go程序的结构是怎样的？

下一篇 06｜构建模式：Go是怎么解决包依赖管理问题的？

精选留言 (16)

写留言

Geek_c1467d
2021-10-22

另外goalng标准布局可以参考下这个：https://github.com/golang-standards/project-layout

作者回复: 这个已经被Go官方否了，https://github.com/golang-standards/project-layout/issues/117#issuecomment-828503689。

5

Bynow
2021-10-23

这部分东西讲解的循序渐进，太棒了，是很多晚上噶好难过没有涉及到的，讲解的可以看出来这这水平和积累，这门课太值了。爱了

展开

作者回复: 感谢支持。你的收获对我来说就是最大的鼓励。

3

swordholder

2021-10-22

这节课的内容非常实用，介绍了最新的最佳实践，大部go语言的书籍都缺少这部分内容。

3

酥宝话不多

2021-10-22

内建函数 make 是第三方构建工具吗 ？

展开

作者回复: make是unix/linux/mac上最常见的第三方构建管理辅助工具。

2

lesserror

2021-10-22

Tony Bail 老师的这一讲关于Go项目的布局标准的讲解非常专业。极客时间孔令飞老师的专栏，对这一布局方式很很好的实践。

有以下疑问，烦请老师抽空解答一下：

...

展开

作者回复: 感谢认真的思考和棒棒的问题，我也认真回答一下:)

1. 是的，如果采用vendor模式，依赖包会缓存在vendor目录下。
2. 在go module机制进入go之前，也就是gopath构建模式时代，我们谈到的所有依赖都是包与包的版本；但go module引入后，所有的版本信息都绑定在module上，所以你在go.mod中看到的require块中的依赖都是module与module的版本，不再是包。
3. 06和07讲会提到。
4. 06,07讲会有例子。

1 1

qinsi
2021-10-22

诶，ESR也写go了？

展开

作者回复: 是的。loccount就是它的作品。他还用go编写了将gcc代码从svn仓库无损(提交历史)地迁移到git的工具。可以看看他切换到go的感悟：https://gitlab.com/esr/reposurgeon/blob/master/GoNotes.adoc

2

Linuaa
2021-10-23

老师可以讲讲 ”Reproducible Build“ 吗，看了一些文章感觉抓不到重点。谢谢老师~

作者回复: 可重现构建，顾名思义，就是针对同一份go module的源码进行构建，不同人，在不同机器(同一架构，比如都是x86-64)，相同os上，在不同时间点都能得到相同的二进制文件。

1

陈东
2021-10-23

知道创世布局，知所以然，才可以更好展开技术工作。

1

浅色幽默
2021-10-22

我去 还没更新完呢 能快点吗

展开

作者回复: 一周三篇，欢迎继续学习。

郑童文
2021-10-22

请问老师go env 里面的配置项究竟是存储在哪儿的？ 网上有说是生成go 命令（go语言的的编译工具）时，直接包含在其中了，也有说是在一个和用户相关的配置文件夹里面，还有的说是来自系统环境变量，那这三种来源的优先级是怎么样的？ 谢谢！

展开

作者回复: 这个问题是不是重复问过，我记得回复过:)

1

Nlife

2021-10-22

老师，这句话"一个 Go 项目里的 internal 目录下的 Go 包，只可以被本项目内部的包导入。项目外部是无法导入这个 internal 目录下面的包的。" 能否再讲解具体一些呢？比如后续我们的课程中是否会讲到这块的实践操作？

展开

作者回复: 举个例子，假设我们有两个go module，两个module的结构如下：

```
.
├── module1
│   ├── go.mod
│   ├── internal
│   │   └── pkga
│   ├── pkg1
│   └── pkg2
└── module2
    ├── go.mod
    └── pkg1
```

module1中的internal/pkga包可以被module1的pkg1和pkg2包所导入。

但无法被module2的pkg1包所导入。

1

2021-10-22

对于最小的布局 我觉的只要这几个文件就好了 main.go. go.mod go.sum. 既然是小项目代码量不多所有代码在 main.go 文件就好。

展开

作者回复: Go语言技术负责人Russ Cox曾谈过这个问题，他认为一个项目的最小布局至少有一个go.mod，一个LICENSE（针对开源项目）。然后就像你说的，在项目根目录下放置go代码即可。对

于tiny项目，一个main.go也是可以的。

1

酥宝话不多

2021-10-22

这里原来是，Makefile

展开

aoe

2021-10-22

看完有点懵，练习一下再说

展开

作者回复: 有问题的话，欢迎留言。

1

罗杰

2021-10-22

对于刚用 Go 开发的时候，凑合能用就行，那个时候一心想着是功能开发，等到功能完成之后，目录结构优化过两三版。感谢老师详细的讲解，对于这种目录结构困惑，我觉得最快找答案的方式就是看优秀的开源库。

展开

作者回复: 历史悠久的优秀开源库，它的布局也在演化。不过有些也没变。

1

Long_hz

2021-10-22

老师你好，请问一下loccount 工具编译的时候缺少go.mod需要怎么解决？

作者回复: loccount只是一个代码统计工具，你可以用其他类似的工具替代。如果非要编译loccount工具，并且它没有go.mod的话，可以下载loccount工具源码后，在你的本地为其创建一个go.mod，然后编译试试。

1